Analysis of a Bootloader in Real Mode Assembly

Introduction

This document provides a detailed explanation of a minimal 16-bit x86 bootloader written in NASM assembly language. The bootloader prints "Hello World!" to the screen using BIOS interrupts and then halts.

Code Listing

```
ORG 0x7c00
BITS 16

start:
    mov si, message
    call print
    jmp $

print:
    mov bx, 0
.loop:
    lodsb
    cmp al, 0
    je .done
    call print_char
    jmp .loop
.done:
    ret

print_char:
    mov ah, 0eh
    int 0x10
    ret

message: db 'Hello World!', 0

times 510-($-$$) db 0
dw 0xAA55
```

Explanation by Sections

1. ORG 0x7C00

The assembler is instructed to assume that the code will be loaded at memory address 0x7C00. This is the conventional address where the BIOS loads the first sector (512 bytes) of a bootable device. This directive is necessary because absolute addresses may be used in the code, and the assembler needs to know the origin.

2. BITS 16

This tells the assembler that the code is intended for 16-bit real mode. Instructions will be encoded using 16-bit operands and addresses. This is the mode the CPU starts in when it first boots.

3. Bootloader Entry Point (start)

The start label marks the beginning of execution. The instruction

```
mov si, message
```

loads the memory address of the string "Hello World!" into the SI register (Source Index), which will be used to read characters one by one. Then

```
call print
```

transfers control to the print routine. After returning from the print function, the instruction

```
jmp $
```

creates an infinite loop. The dollar sign refers to the current address, so this instruction jumps to itself forever. This is necessary to prevent the CPU from executing uninitialized memory, which could lead to undefined behavior.

4. print Routine

The print routine loops through the message and prints each character. The instruction

```
lodsb
```

loads a byte from the address pointed to by SI into the AL register and increments SI. This effectively reads the next character in the string. The next instruction compares AL with zero:

```
cmp al, 0
```

If the null terminator is reached, the function returns. Otherwise, it calls printchar to display the character, then loops.

5. print_char Routine

This routine prints a single character using BIOS interrupt 10h. The instruction

```
mov ah, 0eh
```

sets the function code for BIOS teletype output. The AL register already holds the character from the lodsb instruction. The BIOS interrupt is called:

```
int 0x10
```

This causes the BIOS to print the character in AL to the screen in text mode.

6. message Definition

The message is defined as a sequence of bytes:

```
message: db 'Hello World!', 0
```

This creates a null-terminated string which the print routine will iterate through. The null terminator (0) marks the end of the string.

7. Padding and Boot Signature

A valid bootloader must be exactly 512 bytes long and end with the signature 0xAA55. The instruction

```
times 510-($-$$) db 0
```

fills the space between the end of the program and byte 510 with zeros. The symbols dollar and double dollar represent the current offset and the start of the file, respectively. Finally,

```
dw 0xAA55
```

writes the boot signature. The BIOS checks for this signature at bytes 511 and 512 (last two bytes of the sector). If it is not present, the BIOS will not attempt to boot from the device.

Summary

This bootloader demonstrates the fundamental idea of real-mode programming, including:

- Use of BIOS interrupts for screen output.
- Handling of null-terminated strings in memory.
- Writing position-independent code that fits into 512 bytes.
- Structure and requirements of a boot sector.